Programming the FlexRAM Parallel Intelligent Memory System*

Basilio B. Fraguela Jose Renau† Paul Feautrier‡ David Padua† Josep Torrellas†

Dept. de Electrónica e Sistemas, Universidade da Coruña, Spain
basilio@udc.es

†Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA
{renau,padua,torrellas}@cs.uiuc.edu

‡LIP, Ecole Normale Supérieure de Lyon, France
paul.feautrier@ens-lyon.fr

ABSTRACT

In an intelligent memory architecture, the main memory of a computer is enhanced with many simple processors. The result is a highly-parallel, heterogeneous machine that is able to exploit computation in the main memory. While several instantiations of this architecture have been proposed, the question of how to effectively program them with little effort has remained a major challenge.

In this paper, we show how to effectively hand-program an intelligent memory architecture at a high level and with very modest effort. We use FlexRAM as a prototype architecture. To program it, we propose a family of *high-level compiler directives* inspired by OpenMP called *CFlex*. Such directives enable the processors in memory to execute the program in cooperation with the main processor. In addition, we propose libraries of highly-optimized functions called *Intelligent Memory Operations (IMOs)*. These functions program the processors in memory through CFlex, but make them completely transparent to the programmer. Simulation results show that, with CFlex and IMOs, a server with 64 simple processors in memory runs on average 10 times faster than a conventional server. Moreover, a set of conventional programs with 240 lines on average are transformed into CFlex parallel form with only 7 CFlex directives and 2 additional statements on average.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Intelligent memory architecture, compiler directives, programming heterogeneous computers, parallel languages.

1. INTRODUCTION

The integration of processors and main memory in the same chip is a promising approach to address the processor-memory communication bottleneck [6, 10, 11, 12, 13, 15, 19, 20]. In these chips, processors enjoy short-latency and high-bandwidth communication with memory. One way to use these chips is as intelligent memories that replace all or some of the standard main-memory chips in a server or workstation. This is the approach followed by the Active Pages [15], FlexRAM [10], and DIVA [6] intelligent memory systems.

This use of processor-memory chips as intelligent memory is very appealing because it requires relatively few changes to general-purpose computers, and it supports the execution of applications without modifications. Indeed, applications could be gradually modified or compilers gradually improved to take advantage of intelligent memory capabilities.

Unfortunately, the question of how to effectively program these machines with little effort has remained a major challenge. There are some proposals where the programmer identifies and isolates the code sections to run on the processors in memory [6, 10, 15]. However, these proposals typically require low-level programming. Indeed, the programmer is expected to directly manage low-level operations such as communication via messages, cache management, data layout, or computation and data collocation. As a result, programming these machines is often significantly harder than conventional machines.

An alternative approach has been to use a compiler that automatically partitions the code into loops and other sections and then schedules each section on the appropriate processor [2, 17]. However, this approach has only been

*This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; by the Ministry of Science and Technology of Spain under contract TIC2001-3694-C02-02; and by gifts from IBM and Intel.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPoPP'03, June 11–13, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-588-2/03/0006 ...$5.00.

tried for simple codes or for simple architectures; it has not been shown for a general heterogeneous system with many memory processors and a main processor. Moreover, it is well known that compilers fail to parallelize a wide range of programs.

In this paper, we present a set of compiler directives and necessary operating and run-time system support to hand-program an intelligent memory architecture at a high level and with very modest effort. The resulting applications are also well-tuned and easy to understand and modify. Our proposal is based on *CFlex*, a family of *high-level compiler directives* resembling those of OpenMP [14]. The CFlex environment gives the programmer high-level control over the assignment of computation to the main and memory processors, the layout of the data, and the synchronization between processors in a single-address space. By exposing the high-level architecture to the programmer, it unlocks the performance potential of the system. Moreover, the use of directives makes the programs easy to migrate to other platforms.

Applications can also profit from intelligent memory without the programmer having to be concerned with the architecture organization. This is possible with libraries of highly-optimized functions called *Intelligent Memory Operations (IMOs)*. These functions program the processors in memory using CFlex, but make them transparent to the programmer.

Our discussion and evaluation of CFlex and IMOs are made in the context of FlexRAM [10]. Our simulation results show that, with CFlex and IMOs, a server with 64 simple processors in memory runs on average 10 times faster than a conventional server. Moreover, a set of conventional programs with 240 lines on average are transformed into CFlex parallel form with only 7 CFlex directives and 2 additional statements on average.

The rest of this paper is organized as follows: Section 2 outlines FlexRAM; Section 3 describes the operating and run-time system support; Section 4 describes CFlex; Section 5 presents IMO libraries; Section 6 discusses how CFlex can be applied to other intelligent memory architectures; Section 7 presents the environment that we use to evaluate CFlex and IMOs; Section 8 presents the evaluation; Section 9 discusses related work; and Section 10 concludes.

2. FLEXRAM ARCHITECTURE

A FlexRAM system is an off-the-shelf workstation or server where some of the DRAM chips in the main memory are replaced by FlexRAM processor-memory chips [10]. Each FlexRAM chip contains DRAM memory plus many simple, general-purpose processing elements called *PArrays*. To the main processor of the system, which we call *PHost*, the resulting memory system appears as a versatile accelerator that can off-load memory-intensive or highly-parallel computation. While the machine can have multiple PHosts, in this paper we consider a single PHost system. Each PArray can be programmed independently and, therefore, PArrays can execute MIMD code. PHost and all PArrays share a single address space. Finally, if the PHost runs legacy applications, the memory system appears as plain memory. A FlexRAM system is shown in Figure 1.

The architectural parameters of a FlexRAM chip have been upgraded from [10]; the new parameters are described in [22]. In this upgraded design, each FlexRAM chip has 64

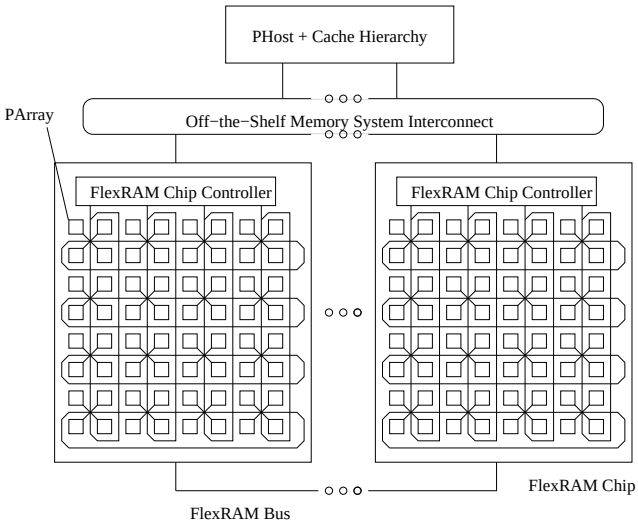

Figure 1. A FlexRAM intelligent memory system.

PArrays, a FlexRAM chip controller (*FXCC*) that interfaces the PArrays to the PHost, and 64 Mbytes of DRAM organized in 64 banks. Each PArray has an 8-Kbyte write-back data cache and a 2-Kbyte instruction cache (Figure 2). A FlexRAM chip has a 2-D torus that connects all the on-chip PArrays. Moreover, all the FlexRAM chips are connected to a communication bus called the FlexRAM bus. With these links, a PArray can access any of the memory banks in its chip and in other FlexRAM chips.

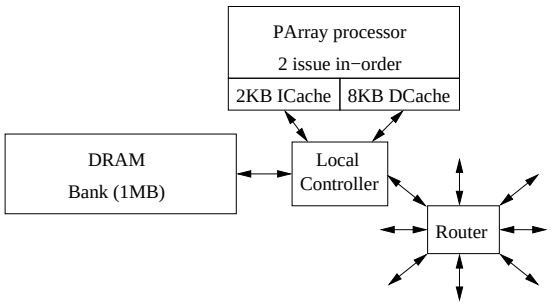

Figure 2. Structure of a PArray and a memory bank inside a FlexRAM chip.

PArrays use virtual addresses in the virtual address space of the application run by the PHost. Each PArray has a small TLB that contains some entries from the PHost's page table. PArrays serve their own TLB misses. However, they invoke the PHost OS for page faults and migrations.

2.1 Interprocessor Communication

Since the memory system interconnect of a FlexRAM machine is off-the-shelf, FlexRAM chips cannot be masters of it. As a result, any communication between PArrays and PHost has to be done in one of two ways: through memory or via memory-mapped registers in the FXCCs. Memory communication involves writing and reading a memory location, and is typically used to pass the input arguments and outputs of the tasks executed on the PArrays. FXCC register communication involves writing and reading special registers in the FXCCs, and is used for commands and ser-

vice requests. We consider FXCC register communication next.

The PHost communicates with the PArrays to spawn tasks on them, answer service requests, and order maintenance operations such as flushing cached pages or invalidating TLB entries. In all cases, the PHost issues a command to an FXCC register and passes two words, namely the address of the routine to execute and a pointer to the input argument buffer. The FXCC stores this information and passes it to the corresponding PArray(s).

A PArray communicates with the PHost to request a service, such as the handling of a page fault. In this case, the PArray writes to a register in its FXCC. The FXCC cannot deliver this request to the PHost because FlexRAM chips cannot be masters of the memory system interconnect. Consequently, the PHost has to periodically poll the FXCCs to check for requests.

2.2 Synchronization

Each FXCC manages a set of locks that can be acquired and released by the PHost and by the on- and off-chip PArrays. Upon a request for a free lock, the FXCC where the lock resides grants ownership to the requester. If the lock is currently taken, the FXCC is able to queue up a certain number of requests, and reply to each of them when it can grant ownership. If the number of requesters exceeds a certain threshold, the FXCC replies back with a "busy" message.

2.3 Data Coherence

A FlexRAM system lacks hardware support for cache coherence between the PHost and PArrays. Data coherence is maintained by explicit (total or partial) cache writeback and invalidation commands [17]. Specifically, before the PHost initiates a PArray task, it writes back the dirty lines in its caches that contain data that may be read by the PArray task. This ensures that the PArray sees the correct version of the data. Moreover, before the PHost executes code that may use results from a task executed by a PArray, the PHost invalidates from its caches the lines with data that may have been updated by the PArray. This ensures that the PHost does not use stale data. As for a PArray, when it completes a task, it writes back its dirty cache lines and invalidates its small cache. The PArray caches also include a dirty bit per word, so that only the modified words actually update memory. This is done to tolerate false sharing between PArrays.

3. OPERATING & RUN-TIME SYSTEM

To use the FlexRAM system, we need several extensions to the OS of the PHost and a small per-PArray kernel. In general, the PHost OS is in charge of all the I/O operations, PHost CPU scheduling, and virtual memory management. The PArray kernel manages the PArray's TLB, and the spawn and termination of local tasks. In addition, we wrote a library-based user-level run-time system for both PHost and PArrays that improves the programmability of the machine. In the following, we describe the management of virtual memory and the user-level run-time system.

3.1 Managing Virtual Memory

PHost and PArrays share a common view of the address space of the tasks that cooperate in the execution of an application. The PArray kernel reads the page table of the process and updates its local TLB. However, only the PHost OS can update the page table, perform page swapping, or migrate pages.

To maximize access locality, the PHost OS tries to map the virtual pages that a PArray references, to physical pages located in the PArray's local memory bank. Currently, we support this approach with a "first-touch" mapping policy, where the first PArray to reference a given page becomes its owner and allocates it locally.

The PHost OS cooperates with the PArray kernels to keep the PArray TLBs coherent when the page table changes. Specifically, there is a shared software structure that contains, for each page mapping, the list of PArrays that cache it in their TLBs. This structure is updated by the PArray kernels as they update their TLBs. Before the PHost OS moves a page to disk, it informs all the PArrays that cache the mapping of that page in their TLBs. Then, the corresponding kernels invalidate that TLB entry and the relevant cache lines. The structure described is also used in other situations when the PHost OS modifies the page table, such as when it migrates pages.

3.2 User-Level Run-Time System

The PHost and PArray user-level run-time system performs task management, synchronization, heap memory management, and polling. We consider these functions in turn.

Tasks are spawned by the user-level run-time system. Each task has an associated software buffer that contains the task input data that is not shared and, sometimes, the results of the task. The former includes private copies of global variables that the task needs; the latter is not necessary if all the results are stored in shared memory structures. To start a task in a PArray, the PHost run-time system fills the inputs in this buffer and then sends a spawn message to the FXCC of the FlexRAM chip where the task is to run. The PHost can then use the run-time system to spin on a location in the buffer that the PArray task will set when it finishes. PArrays cannot spawn tasks.

The run-time system also includes synchronization routines that use the FXCC locks, often building higher-level constructs such as barriers.

The run-time system also performs heap memory management, including parallel allocation and deallocation of heap space by the PArray tasks. Standard `malloc` and `free` functions are used, both in the PHost and PArrays. Those in the PArrays operate on local pages of the heap that the PHost run-time system has allocated, assigned to that particular PArray, and requested that the OS map them in that particular PArray's local memory bank. When the PArray task runs out of local heap, it asks for more to the PHost run-time system.

Finally, another function of the run-time system in the PHost is to poll the FXCCs to detect PArray requests.

Note that the run-time system exports its functions and variables not only to the compiler but also to the programmer. The complete description of its interface is found in [5].

4. CFLEX

To exploit the functionality of FlexRAM while keeping the portability of the programs, we program the FlexRAM system using a family of compiler directives. Our computational model requires these directives to be able to express

the task partitioning between the PHost and the PArrays, and the synchronization between the generated tasks. Other desirable properties are scalability as the number of processors available in memory increases, and hiding as much system details as possible from the programmer. It is also important that the directives be powerful and flexible, so that they allow the generation of parallel programs that are easy to read and modify for a wide class of problems.

The last objective is essential for our environment because of the nature of the applications that are most suitable for our kind of architecture. Specifically, the PArrays are much simpler than the PHost, have a lower clock rate, and lack floating-point units. This means that they are not particularly well suited to speed up typical parallel applications based on floating-point operations and loops that use regular, cache-friendly data structures. Instead, the PArrays' most valuable property is their low memory access time, which allows them to accelerate irregular, memory-intensive applications. This is an interesting challenge for the design of our family of directives. Indeed, currently available parallelization directives such as OpenMP [14] or HPF [8] are especially oriented to loop-parallel codes, and are not well suited for irregular applications like those using pointers or indirect accesses.

CFlex is a family of directives that addresses these issues. The structure of CFlex directives is similar to that of OpenMP directives. We implemented CFlex as annotations to the C language, and use C in the discussion below. However, CFlex can be easily used with other languages like FORTRAN.

The general structure of a CFlex directive is

#pragma FlexRAM *directive-type [clauses]*

CFlex directives may be classified in three groups:

- *Execution modifiers* indicate how a given segment of code should be executed. These are the most widely used directives. They include the requests to spawn a given portion of the program for execution on a given processor or group of processors.

- *Data modifiers* request that data structures satisfy certain conditions. An example are directives to pad one of the dimensions of an array to make its size a multiple of the page size.

- *Executable directives* are instructions that must be executed by the parallel program. Examples of this class include barriers or prefetch operations.

In the following, we briefly discuss the three kinds of directives and then show some examples of their use. A complete description of CFlex can be found in [5].

4.1 Execution Modifiers

These directives partition the computation into PHost and PArray tasks, and synchronize tasks. The most important execution modifier directives are `phost` and `parray`, which specify the kind of processor where the statements that immediately follow them should be executed. Since PArrays cannot spawn new tasks, these directives may only appear in the code executed by the PHost, and they will spawn either a new PHost task or a PArray task. A series of optional *clauses* enrich the semantics of the directive:

- `on_home(`*x*`)` specifies that the task should be executed on the PArray whose local memory bank contains the x data structure. If the *directive-type* is `phost`, then this clause only has effect when the target computer is a NUMA machine. In this case, it specifies that the new task should be run on the PHost whose local memory contains x.

- `sync`/`async` specifies whether the task spawning the new task must stop until the new task finishes (`sync`) or it can continue (`async`). The default is `sync`. A task does not finish until all the tasks that it has spawned have finished. CFlex only allows the creation of asynchronous tasks inside the syntactical scope of a synchronous task. The end of a `sync` task T can therefore be used as a synchronization point of all the tasks spawned inside T. This approach is powerful, flexible, and simple as will be illustrated in Section 4.4.

- `if(`*cond*`)` controls the execution of the directive where it appears. The directive is executed only if *cond* is true.

- `else` also controls the execution of a directive. The directive is executed if the *cond* in the `if` clause in the immediately preceding directive is false. Note that execution modifiers cannot be nested for the very same piece of code. Therefore, there is no need to implement symmetric conditionals.

- `shared`, `private`, `lastprivate`, `firstprivate`, `reduction` have the same semantics as the directives of the same name in OpenMP. In contrast to OpenMP, CFlex allows them to apply to only one part of a structure, such as one field of a struct or one element of a vector.

- `flush` specifies variables such that, if their corresponding lines are dirty in the PHost cache, the lines are written back to memory. This is done so that the PArray(s) executing the task(s) have access to the latest version of the data they require in memory. If the task is to run on a PArray and this clause is not present, the compiler writes back to memory all the dirty lines in the PHost cache. The programmer can use this clause to restrict the writeback to a certain set of variables.

Note that the ability to spawn new PHost tasks allows CFlex to express parallelism in systems without processing in memory. In this case, the `on_home` clause is useful to control locality of execution when the PHosts belong to a NUMA machine.

A `migrate` clause could be added in the future to designate shared data structures whose pages should migrate to the first PArray or PHost that touches them after the task(s) created by this directive begin their execution. However, page migration can be very costly.

A third execution modifier is `critical`, which declares a code segment as a critical section with a given name. All the critical sections with the same name are mutually exclusive. This is accomplished through the use of the same FXCC lock for all of them. This directive can take the single optional clause `flushinval` that specifies the variables that may be written within the critical section, or read in the critical section and written somewhere else. In this case, the cache lines with such variables are written back from the

processor's cache (if dirty) and invalidated before entering the critical section. This forces the processor to read the latest version of the variables in the critical section. Moreover, these cache lines are written back to memory (if dirty) when the processor exits the section. This enables the next processor that will enter the critical section to access the new version of the variables. If the `flushinval` clause is not present, the operations described are performed on the whole cache.

4.2 Data Modifiers

We propose three directives of this kind, namely `alignable`, `page_aligned`, and `align`. The first two directives can precede the declaration of a C `struct` or `union` and instruct the compiler to pad the data structure to align it. Specifically, the `alignable` directive increases the size of the data structure until it becomes a power of two; the `page_aligned` directive increases the size until it becomes a multiple of the page size. Finally, the `align(`*array_name*`[][]...)` directive aligns a dimension of the array *array_name* to a page boundary. The dimension is specified by the number of square-bracket pairs in the directive.

4.3 Executable Directives

CFlex has several executable directives that perform a variety of functions. As an example, the `barrier` directive implements a barrier for n processors using FXCC locks. As another example, the `flush` directive specifies a series of variables to write back to memory if the corresponding lines are dirty in the cache of the processor. This directive takes the optional `invalidate` clause, which additionally causes these lines (or a subset of them) to be invalidated after the potential writeback.

4.4 Examples

To gain more insight into the CFlex directives, we now show several simple examples of their use. A first example involves traversing a linked list and performing some processing on each of its nodes in parallel (Figure 3). The first directive in the figure generates a synchronous (`sync`) task in the PHost that will execute a whole loop. Note that generating a synchronous task in the same processor has no overhead. The purpose of this directive is simply to provide a context for synchronization of the tasks that are spawned by each iteration the loop. Each one of these tasks executes on the PArray whose local memory bank contains `p->data`.

```
#pragma FlexRAM phost sync
  for(p = head; p != NULL; p = p->next)
#pragma FlexRAM parray async on_home(*(p->data)) \
                             firstprivate(p)
    process(p->data);
```

Figure 3. Parallelized linked-list processing using the `sync` and `async` clauses.

Each one of these asynchronous tasks receives a privatized copy of the value of pointer `p` for the corresponding iteration, and processes a node from the list. Since these tasks are asynchronous, the PHost task continues to iterate the loop and spawn tasks until it reaches the end of the loop. Then, the synchronous task waits for all of the asynchronous (`async`) tasks to complete. When they do, the synchronous task finally completes.

In this example, we have illustrated the general scheme used to parallelize a loop: declare the loop as a synchronous task and each of its iterations as an asynchronous one. Since loops are the most common source of parallelism, we have extended CFlex with a `pfor` clause that tells the compiler to break the loop following it into a series of asynchronous tasks and wait for their completion before continuing. Thus, the previous loop can be re-written using a `pfor` clause as shown in Figure 4. Note that although OpenMP is largely designed for loop parallelism, a parallel version of this loop in OpenMP would require the use of the `ordered` clause and/or a rearrangement of the code, which would be less readable and efficient.

```
#pragma FlexRAM parray pfor on_home(*(p->data)) \
                            firstprivate(p)
  for(p = head; p != NULL; p = p->next)
    process(p->data);
```

Figure 4. Parallelized linked-list processing using the `pfor` clause.

A more complex parallelization scheme is required when there are portions of code that must be run sometimes in the PHost and sometimes in the PArrays. An example is shown in Figure 5. The example implements the routine that allocates the tree in the TreeAdd application from the Olden suite [16]. The routine allocates a binary tree of height `level`. Our parallelization strategy selects a level `cutlevel`. The nodes below that level are allocated by the PArrays and the nodes in that level and up to the root (highest level) are allocated by the PHost. Recall that the runtime system allows the PArrays to allocate and deallocate heap memory in parallel (Section 3.2).

```
tree_t *TreeAlloc (int level) {
  if (level == 0) return NULL;
  else {
    struct tree *new, *right, *left;

    new = (struct tree *) malloc(sizeof(tree_t));
#pragma FlexRAM phost if (level >= cutlevel)
    {
#pragma FlexRAM parray async if (level == cutlevel)
#pragma FlexRAM phost async else
      left = TreeAlloc(level-1);

#pragma FlexRAM parray async if (level == cutlevel)
      right=TreeAlloc(level-1);
    }
    new->val = 1;
    new->left = (struct tree *) left;
    new->right = (struct tree *) right;
    return new;
  }
}
```

Figure 5. Parallelized tree allocation.

The tree is allocated top down starting at the root. As long as the `level == cutlevel` condition is not satisfied, `TreeAlloc` is called by the PHost. Under these conditions, however, the PHost spawns a new task on the PHost to build the left subtree of each node. We follow the approach of creating multiple PHost tasks to avoid modifying the original code. When `cutlevel` is reached, both the left and right subtree allocation tasks are run on PArrays. Note that a

single PArray task allocates a whole subtree. This is because PArrays ignore the `phost` and `parray` directives, since they cannot spawn new tasks. With this support, a whole subtree is allocated in a local memory bank. The resulting partitioning of the work is shown in Figure 6. In the figure, the dashed lines represent the spawn of a new task on the PHost. If there is not enough space in a memory bank to keep a whole subtree, some pages are allocated from another bank, and the PArray accesses them remotely. To avoid these remote accesses, it is better to choose a `cutlevel` that guarantees that a subtree built by a PArray fits in its local memory bank.

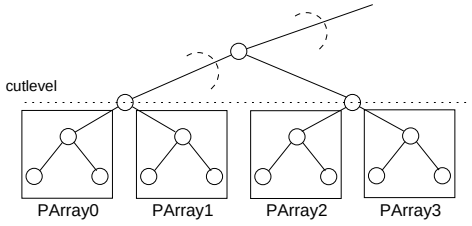

Figure 6. Partition of the work between the PHost (level `cutlevel` and above) and the PArrays for the code in Figure 5.

This work-partitioning strategy may be applied to any parallel processing of tree data structures. For example, the second step of the TreeAdd application is a reduction that adds the values stored in all the nodes of the tree; we have parallelized it in the same way as the allocation. As an additional optimization, the task that adds the values in a subtree built by a PArray is spawned using the `on_home` clause to ensure that the reduction is performed by the PArray that owns the subtree. This approach exploits locality.

Our compiler generates two versions of the routine in Figure 5, one for the PHost and one for the PArrays. While the PHost version includes the code associated with the directives, the PArray version does not. The reason is that PArrays cannot create new tasks, so these directives do not apply. Recall also that PHost and PArrays have different ISAs. Consequently, the back-end compiler has to generate two versions anyway, even if the high-level code is exactly the same for both kinds of processors.

Finally, we consider an example where several data structures need to be processed together. In this case, while we can use the `on_home` clause to ensure that accesses to one data structure are local, the accesses to the other data structures may end up being remote. To address this problem, we could use a `migrate` clause to migrate the pages of the remote data structures, but the overhead could be high. Instead, an approach that often works is to make use of the first-touch page-placement policy of our OS. With this support, we may implicitly align at the page level the data structures that are used together throughout the code.

As an example, Figure 7 shows two loops in the Swim application from the SPEC OMP2001 suite. In the figure we use the FORTRAN form of the CFlex directives. The first loop is from the `INITAL` routine and contains the first accesses in the program to vectors `UOLD`, `VOLD` and `POLD`. The `on_home` clause forces iteration `J` to be executed by the PArray that holds element `(1,J)` of matrix `U`. This PArray is also the first processor in the system to access column `J` of `UOLD`, `VOLD`, and `POLD`. As a result of our first-touch policy, the OS places the pages that contain such columns in the local memory bank of the same PArray. The same strategy was used in previous loops to place the pages containing the columns of matrices `U`, `V`, and `P`. If the column sizes are (or can be made) such that the columns can be aligned to page boundaries, and each matrix starts at a page boundary, then all the accesses in the code end up being local. Figure 8 shows the resulting data layout assuming that each memory bank ends up allocating 8 columns from every matrix. In the figure, each box inside a memory bank represents a page-aligned chunk of memory that extends over several pages.

```
C$FlexRAM parray pfor on_home(U(1,J)) private(I)
      DO J=1, NP1
        DO I=1, MP1
          UOLD(I,J) = U(I,J)
          VOLD(I,J) = V(I,J)
          POLD(I,J) = P(I,J)
        END DO
      END DO
      ...

C$FlexRAM parray pfor on_home(U(1,J)) private(I)
      DO J=1,N
        DO I=1,M
          UOLD(I,J) = U(I,J) + ALPHA * ...
          VOLD(I,J) = V(I,J) + ALPHA * ...
          POLD(I,J) = P(I,J) + ALPHA * ...
          U(I,J) = UNEW(I,J)
          V(I,J) = VNEW(I,J)
          P(I,J) = PNEW(I,J)
        END DO
      END DO
```

Figure 7. Alignment of data structures using the `on_home` clause and the first-touch page-placement policy.

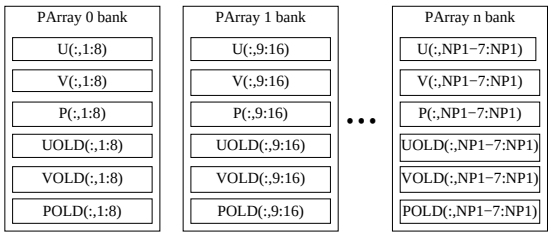

Figure 8. Alignment of the matrix columns from the first loop in Figure 7. In the figure, each box inside a memory bank represents a page-aligned chunk of memory that extends over several pages.

When related data structures are referenced later in the code, we have to distribute the corresponding loop among the PArrays following the same policy. This is illustrated in the second loop of Figure 7, taken from the `CALC3` routine.

Overall, note that none of the examples in this section required adding or modifying *any line* of the original sequential code. All the parallelization semantics have been expressed by means of compiler directives. This is fairly typical of the codes that we have parallelized using CFlex.

5. INTELLIGENT MEMORY OPERATIONS

The family of compiler directives presented in the previous section enables the parallelization of a large set of codes and

Function Description	Abstract Expression	Syntax
Apply func `f` with arg `a`	exec `f(v(i),a)`, $0 \leq$`i`$< N$	`Vector_apply(v,f,a)`
Search element that fulfills condition `f` with arg `a`	ret any `v(i)` such that `f(v(i),a)`$\neq 0$, $0 \leq$`i`$< N$	`Vector_search(v,f,a)`
Generate vector with the result of applying func `f` with arg `a`	`v2(i)=f(v(i),a)` $0 \leq$`i`$< N$	`v2=Vector_map(v,f,a)`
Reduce vector applying func `f`, whose neutral element is `a`	ret `f(...f(f(a,v(0)),v(1))...)` where `f(a,x)=x`	`Vector_reduce(v,f,a)`
Process together two vectors and an arg `a`, generating a new vector	`v3(i)=f(v(i),v2(i),a)` $0 \leq$`i`$< N$	`v3=Vector_map2(v,v2,f,a)`

Table 1. Examples of IMOs for vector containers.

the exploitation of locality while hiding many of the system details from the programmer. However, to develop efficient programs, the programmer must be aware of the existence of the FlexRAM chips and must decide how to partition and coordinate the work between the PHost and the PArrays.

All these details can be hidden and performance can be improved with the use of library routines. Specifically, we propose the use of a library of Intelligent Memory Operations (IMOs). IMOs are encapsulated operations that make use of the PArrays. IMOs perform common operations on data structures that are often used in programs.

Simple examples of IMOs are finding the minimum value in a vector of numbers or adding two matrices. Other, more structured examples of IMOs are STL classes [18]. Such IMOs may define and operate on containers such as vectors, lists, hash tables, or sets. They may make use of the intelligent memory to perform parallel allocations and deallocations, searches, insertions, retrievals, and other computations on the elements in the containers. Some examples of IMOs for vector containers are proposed in Table 1.

To make code containing IMO calls portable to conventional machines, IMO libraries implement two versions of the functions provided to the programmer: one to be used when no FlexRAM chips are detected in the system, and another one programmed in CFlex that exploits the capabilities of the FlexRAM chips. Both versions should be highly optimized and completely hide from the programmer issues such as task partitioning, scheduling, and synchronization.

6. OTHER ARCHITECTURES

There are other architectures that, like FlexRAM, have many simple processors in main memory that cooperate with the main processor. Examples of such architectures are Active Pages [15] and DIVA [6]. For these architectures, CFlex and IMOs can provide appropriate programming support.

CFlex provides a solution to the computational issues involved in the programming of Active Pages [15]. These issues are: partitioning between processor and memory, coordination, computational scaling with the number of memory processors and their associated memory, and data manipulation. The last issue is particularly related to the use of IMOs. In fact, [15] illustrates the advantages of Active Pages for data manipulation using an implementation of the STL array class that exploits Active Page functions, very much in the line of our IMOs.

DIVA [6] can also be programmed using explicitly-parallel programming languages that permit some programmer control of locality, as is the case for CFlex. Besides, like FlexRAM, DIVA also needs support from the language/compiler to keep the host cache coherent with the memory processors using write-backs and invalidations, although the processors in memory lack data cache.

However, there are differences that seem to make Active Pages and DIVA more sensitive than FlexRAM to the data placement. Thus, extensions of CFlex to specify placement and alignment of data structures would be very useful for both architectures when the implicit alignment used in this paper is not enough. In the case of Active Pages, the problem is that all the communications between the memory processors are serialized through the PHost. As a result, the PHost may become a bottleneck when poor data placement results in many non-local accesses.

In DIVA, the exchange of data between memory chips requires the use of messages called parcels. Passing a parcel involves software processing by either user- or supervisor-level code at both ends [7], which makes it slower than the hardware approach followed by FlexRAM. Consequently, it would also be important for a CFlex programmer to maximize locality. Moreover, a CFlex compiler may improve performance by inserting efficient code required to manage the message passing when the communication between tasks is regular enough. If this is not the case, an implicit mechanism such as an interrupt with its service routine, could be provided by the run-time system to request non-local data each time that it is needed.

7. EVALUATION ENVIRONMENT

For our evaluation, we use an execution-driven simulation infrastructure that can model aggressive out-of-order superscalar processors and complete memory subsystems.

7.1 Architecture Modeled

Our baseline architecture is a workstation with a high-performance 1.6 GHz five-issue PHost processor similar to IBM's Power4 [3]. The performance of this workstation is compared to that of two upgraded versions of it: one where the plain main memory is replaced by a single FlexRAM chip, and one where it is replaced by two FlexRAM chips.

The main architectural parameters of the system modeled are shown in Table 2. In the table, the times for each processor are measured in terms of that processor's cycles. Note that the PHost is very powerful, has a large L2 cache, and is able to sustain many simultaneous memory accesses. Consequently, the baseline architecture is very aggressive. On the other hand, recent advances in merged logic-DRAM technology have enabled the integration of high-speed logic with high-density memory in the same chip [9]. Consequently, we have set the frequency of the PArrays to be 75% of that of the PHost. Recall that each FlexRAM chip has 64 PArrays. As shown in Table 2, these PArrays are much simpler than

the PHost. Specifically, each PArray has a single integer adder and shares an integer multiplier with 3 other PArrays. Moreover, PArrays lack floating-point hardware, which they emulate in software. We assume that the latencies of emulating a floating-point add/subtract, multiply, and divide/sqrt operation are 3, 10, and 80 cycles, respectively.

PHost Processor	PHost Caches	Bus & Memory
Freq: 1.6 GHz	L1 Size: 32 KB	Bus: Split Trans
Issue Width: 5	L1 OC,RT: 1,3	Bus Width: 8 B
Dyn Issue: yes	L1 Assoc: 2	Bus Freq: 400 MHz
I-Window Size: 64	L1 Line: 128 B	Mem RT: 180 (112.5 ns)
Ld,St Units: 2,1	L1 MSHR: 16	
Int,FP Units: 5,4	L2 Size: 1 MB	
Ld Queue: 32	L2 OC,RT: 4,12	
St Queue: 32	L2 Assoc: 8	
BR Penalty: 12	L2 Line: 128 B	
TLB Entries: 128	L2 MSHR: 8	
PArray Processor	**PArray Cache**	**FlexRAM Torus, Bus**
Freq: 1.2 GHz	L1 Size: 8 KB	Avg Torus RT: 14
Issue Width: 2	L1 OC,RT: 1,2	Torus Freq: 1.2 GHz
Dyn Issue: no	L1 Assoc: 2	Bus: Split Trans
Ld,St Units: 1,1	L1 Line: 32 B	Bus Width: 8 B
Int,FP Units: 1,0	Blocking	Bus Freq: 400 MHz
Ld,St Queue: 2,2		
BR Penalty: 6		
TLB Entries: 32		
PArrays/Chip: 64		

Table 2. Parameters of the architecture modeled. In the table, BR, OC, RT, and MSHR stand for branch, occupancy, latency of a round trip from the processor, and miss status handling register, respectively. Each PArray has a single integer adder and shares an integer multiplier with 3 other PArrays.

We also simulate the parts of the OS and run-time system that are most likely to be exercised. For the OS, this includes building and keeping a two-level page table, allocating physical pages, mapping virtual pages to physical ones, maintaining the TLBs in both PHost and PArrays, and performing task scheduling. As for the run-time system, we model it completely, including task spawning, memory allocation by both the PHost and PArrays, and periodic polls and other accesses to synchronize PHost and PArrays. The size of the pages used is 16 Kbytes.

7.2 Applications Used

To evaluate the programmability and performance of a FlexRAM system using CFlex or IMOs, we select eight applications. These applications have a wide variety of characteristics, which can help us discover which attributes are most suitable for FlexRAM, and what problems arise in FlexRAM programming. These applications have been annotated by hand with CFlex directives or with calls to an IMO library that we created. We have also modified the SUIF compiler [21] to accept CFlex pragmas and compile the applications. The resulting executable file is passed to our simulator infrastructure.

The IMO library contains operations to handle singly-linked lists. It includes both STL-like operations such as insertion, retrieval, or search, and many high-level operations. Examples of the latter include those in Table 1 applied to linked lists. Another example is to process with a function all the pairs of elements taken from two lists. The library uses a linked list that is distributed across the PArrays. It makes extensive use of the run-time system functionality that enables PArrays to allocate and deallocate portions of heap memory in parallel. All the operations are used in our applications. Overall, the library contains 714 lines of source code, including the CFlex directives.

Table 3 lists the applications used. TSP and TreeAdd are taken from the Olden suite of pointer-intensive sequential applications [16], Swim and Mgrid are from the SPEC OMP2001 suite, Dmxdm and Spmxv are numerical kernels, and Distance and Path were written from problem descriptions in [4]. Distance and Path are coded with IMO library calls, while the other applications use CFlex directives.

We use four axes to broadly classify the behavior of each application. Specifically, the access patterns may be irregular due to pointers (*Ptr*) or due indirections in the form of subscripted indices (*Ind*), or regular (*Reg*). The computation may use mostly integer (*Int*) or floating-point (*FP*) operations. When several data structures are involved in the computation, we are able to align all of them (*Yes*), only some of them (*Part*), or none of them (*No*). Finally, the typical number of instructions in the tasks sent to the PArrays may be tens of thousands (*Small*), hundreds of thousands (*Med*), or over one million (*Large*). The tasks in Mgrid have a variety of sizes. Overall, we can see that our applications cover a wide variety of behaviors.

The table also lists the data set size of the applications, and the average Instructions Per Cycle (IPC) of the applications running on the architecture without FlexRAM chips.

The last three columns of the table attempt to estimate the effort required to map the applications to the FlexRAM system. From left to right, they list the original number of lines of code, the number of CFlex directives inserted, and the additional lines required to map the code to the FlexRAM system. For the two applications coded with IMO calls, the last two columns have a slightly different meaning: number of CFlex directives in the IMO functions used, and the static number of calls to IMO functions, respectively. The original code size reflects the number of lines of the applications without including the IMO library. Note that rather than attempting to heavily modify the code in order to exploit as much parallelism as possible, we have focused on the simplicity of the mapping process by making as few modifications as possible, as the figures in the table show. For example, the six applications with CFlex directives originally have 240 lines of code on average, and are transformed into CFlex parallel form with only 7 CFlex directives and 2 additional statements on average.

7.2.1 Details on Individual Applications

To help understand the mapping better, we now give some details on individual applications. We start with TSP and TreeAdd, which operate on trees built with pointers. TreeAdd is parallelized and mapped as discussed in Section 4.4. TSP follows a similar approach but has some differences. Specifically, subtrees in TreeAdd are processed independently by different PArrays, while the PHost performs the computation above a certain tree level. In TSP, instead, processing a node of the tree requires accessing the whole subtree below the node. Thus, assigning the processing of the upper levels of the tree to the PHost would leave much parallelism unexploited. In our parallelization, we let PArrays work at

Applic.	Coding	Application Characteristics				Data Set	Baseline	Original	Number	Additional
		Access	Data	Align	Task Size	Size (MB)	IPC	Lines	Directives	Lines
TSP	CFlex	Ptr	FP	-	Large	22	0.85	485	12	5
TreeAdd	CFlex	Ptr	Int	-	Large	40	1.01	71	8	4
Swim	CFlex	Reg	FP	Yes	Med	28	0.95	272	8	0
Mgrid	CFlex	Reg	FP	Yes	Var	55	2.35	470	13	0
Dmxdm	CFlex	Reg	FP	Part	Large	23	3.47	81	1	2
Spmxv	CFlex	Ind	FP	No	Med	36	0.59	47	1	0
Distance	IMOs	Ptr	Int	-	Large	1	2.04	108	17	7
Path	IMOs	Ptr	Int	-	Small	13	0.33	165	17	9
Average						27.3	1.45	212.4	9.6	3.4

Table 3. Characteristics of the applications used.

all levels of the tree and only reserve the root for the Phost. As the processing moves up the tree, there are fewer subtrees, which means both fewer active PArrays and that those PArrays have to access data in more memory banks.

Swim and Mgrid use the *test* input data set. Their access patterns are very regular and floating-point operations dominate the computation. We exploit the vast loop-level parallelism in these applications by simply replacing the original OpenMP directives by the corresponding CFlex ones. Note that, although the data set size of Swim uses about 28 Mbytes, its memory footprint is larger than 64 Mbytes because of internal page fragmentation. As a result, Swim requires at least two FlexRAM chips to execute without intensive swapping. Consequently, we did not perform experiments with Swim using a single FlexRAM chip.

Dmxdm and Spmxv are floating-point kernels. Dmxdm multiplies two 1000×1000 dense matrices with blocking in the three dimensions. Each submatrix is copied into consecutive locations to improve the locality. One of the loops is also unrolled and jammed. Spmxv multiplies a sparse 10000×10000 matrix with three million entries to a vector. The matrix is stored in Compressed Row Storage (CRS) format [1]. Column indices and non-zero values are not aligned because of their different size (four and eight bytes, respectively). These two kernels are parallelized by assigning a different PArray to compute a block of rows of the destination matrix (in Dmxdm) or a set of consecutive elements of the destination vector (in Spmxv). In both kernels, a single directive is required to parallelize the outer loop.

Finally, Distance and Path work with singly-linked lists using our IMO library of operations on these data structures. Distance takes a set of points in a two-dimensional space and finds all pairs of points that are closer than given distance; Path finds the shortest path between two given points in a graph. The IMO functions are designed to be very efficient for both the sequential (no FlexRAM) and the parallel (FlexRAM) execution of these applications. Still, there are some cases where the performance of the sequential execution can be hindered by IMO code structure that is better suited for parallel execution. In those cases, we wrote versions of these IMO functions that are optimized for the sequential execution, and we use them when evaluating the no-FlexRAM architecture.

8. EVALUATION

To evaluate the impact of the intelligent memory, we measured the execution speedups of the FlexRAM system over the baseline workstation. In the following, we first examine the initial speedups obtained, and then evaluate compiler, run-time, and hardware optimizations.

8.1 Application Speedups

In our evaluation, we examine FlexRAM systems with one or two FlexRAM chips. Figure 9 shows the resulting speedups for each application and their geometric mean (GMean). Recall that, to accommodate the working set of Swim, we need two FlexRAM chips. In the figure, the speedups correspond to the execution of the complete applications. All applications spend more than 99% of their original execution time in the section of the code parallelized with CFlex.

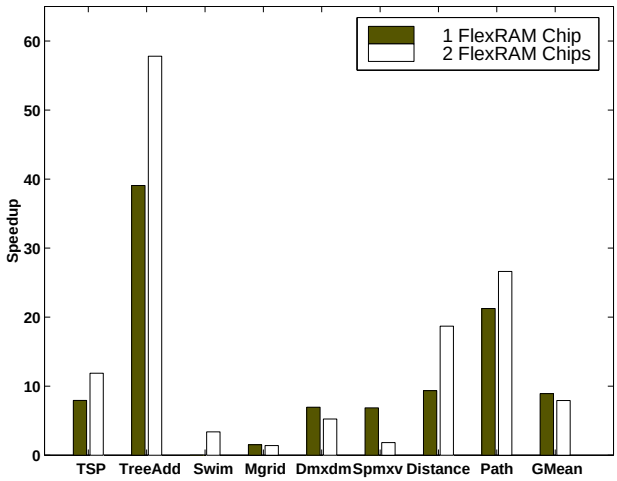

Figure 9. Execution speedups obtained using the FlexRAM system.

The figure shows that, for one FlexRAM chip, the speedup figures are quite good: while there is significant variability across applications, the geometric mean is about 9. In general, the applications with the highest speedups are those with irregular access patterns and those with integer computation. This was to be expected because dense numerical applications tend to make better use of large caches and good floating-point support of the PHost-only baseline workstation. In addition, applications such as Path, where PArray tasks largely use data located in the local memory bank, obtain better speedups than applications such as Spmxv where PArrays require data from other banks.

The locality of PArray accesses also affects the changes

in speedups as we go from one to two FlexRAM chips. In applications with good locality, the speedups go up, while in those with poor locality, the opposite occurs. In the second class of applications, the FlexRAM bus becomes a bottleneck for accesses to banks in other chips. Overall, without considering the contribution of Swim, the geometric mean of the speedups for two FlexRAM chips is not higher than for one. In general, contention in the FlexRAM bus and overheads due to synchronization and task spawning will grow with the number of FlexRAM chips. Consequently, unless the application requires little data movement and synchronization, it is generally advisable to use the smallest number of FlexRAM chips required to hold its data set.

8.2 Compiler and Run-Time Optimizations

The CFlex versions of our applications often use `on_home` clauses to leverage our first-touch page allocation policy and align data structures for local computation. We call these versions *Opt*, and have used them to calculate the speedups shown in Figure 9. Surprisingly, we found that CFlex versions of Swim and Mgrid without any `on_home` clauses are faster than their corresponding *Opt* versions. In these new versions, which we call *NoOpt*, tasks are scheduled following the default policy when the `on_home` clause is missing: cyclically across chips and then, within a chip, cyclically across PArrays. Pages are still allocated using the first-touch policy. The difference in speedups between the *NoOpt* and *Opt* versions is shown in the first two bars of Figure 10. Note that this effect only occurs for Swim and Mgrid.

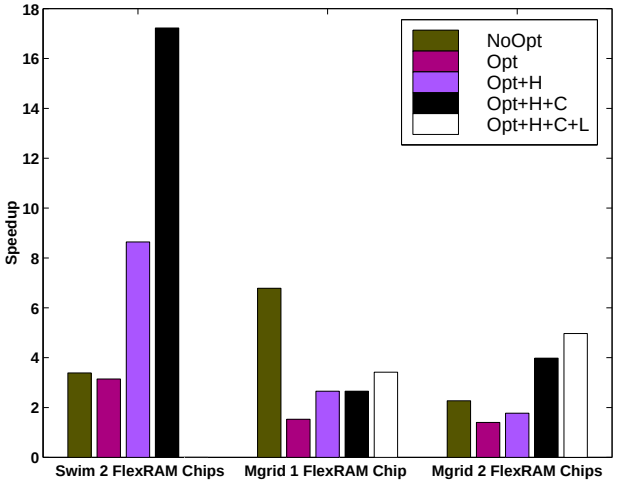

Figure 10. Impact of compiler and run-time optimizations for task spawning and mapping in Swim and Mgrid.

These experiments helped us identify at least three compiler or run-time inefficiencies in spawning and mapping tasks. The first inefficiency results from the way tasks are spawned under the `on_home` clause. Therefore, it only affects the *Opt* version. It occurs when the computation segments to be assigned to consecutive tasks access consecutive pages in the memory bank of the same PArray. The compiler generates one task for each page and assigns all the tasks to the same PArray. Unfortunately, the creation of so many tasks causes significant spawning and synchronization overheads. Moreover, assigning them all to the same PArray may reduce parallelism. The reason is that our run-time system can only spawn a task when all the previous tasks generated by the PHost have already been spawned. Moreover, if the destination PArray is busy executing another task, the spawn request is queued up in a register of the chip's FXCC. If the FXCC runs out of registers, the spawn request cannot be queued up and the run-time system has to wait for tasks to finish.

To eliminate this inefficiency, we changed our compiler as follows. Instead of generating different tasks that access consecutive pages of the same memory bank, the compiler combines all the work into a single task. This approach eliminates overheads and the potential run-time stall problem mentioned above. We call this optimization *H* for *Home-allocation*, and apply it to the *Opt* versions to obtain *Opt+H*. Figure 10 shows that *Opt+H* delivers higher speedups, especially for Swim.

The second inefficiency occurs in machines with more than one FlexRAM chip when tasks are mapped without the `on_home` clause. Recall that, in this case, tasks are mapped cyclically among the FlexRAM chips. This is done for two reasons: to balance the usage of the chips and to reduce the likelihood of run-time stall due to running out of FXCC registers. Unfortunately, consecutive task spawns generate tasks that usually access related pieces of data, and often share the same data. Spawning these tasks on different chips often causes our first-touch page allocator to map the pages of consecutive portions of vectors and arrays on different chips. As a result, if tasks want to access information that is near in the virtual space, they are forced to use the FlexRAM bus and go across chips. Note that this problem still affects our *Opt* versions, since some of their code sections do not use the `on_home` clause.

To eliminate this inefficiency, we change the mapping policy for consecutive tasks when the `on_home` clause is not present. We perform cyclic mapping of tasks within a chip before moving to mapping tasks to the next chip. Consequently, each chunk of 64 consecutive tasks is mapped in the same chip. We call this optimization *C* for *Consecutive-on-chip*, and apply it to the *Opt+H* versions to obtain *Opt+H+C*. Figure 10 shows the resulting speedups, which are now significantly higher. Note that *Opt+H+C* does not apply to single-chip systems.

Finally, there is a third, potential inefficiency that is intrinsic to the use of the `on_home` clause. When assigning the computation to the PArray(s) on whose bank the data is located, we certainly obtain better locality. Unfortunately, we also restrict the number of PArrays that cooperate in the computation and, therefore, restrict parallelism.

Addressing this inefficiency involves distributing the data among as many PArrays as possible, which has negative effects on locality. Besides, we are limited by the fact that the granularity of the distribution in our system is the page. Therefore, computation to be parallelized using the `on_home` clause that operates on small pieces of data can only be split among a few PArrays. This effect hurts Mgrid's performance. Consequently, we optimize task spawning and mapping as follows: `on_home` clauses are applied only on loops that have more than 60 iterations. Otherwise, the loop is parallelized without applying the clause, thus losing locality but gaining in parallelism. This optimization attempts to ensure that the number of PArrays that execute the loop iterations is not too small despite the restriction that `on_home` imposes.

We call this optimization L, for *Limited on_home*. This optimization is only needed in Mgrid. The last bar in Figure 10 shows the small improvements obtained when applying it to the *Opt+H+C* version of Mgrid, to obtain version *Opt+H+C+L*.

8.3 Hardware Optimizations

In several applications, the PHost spawns many small tasks. Unfortunately, task spawning has overhead, as the PHost first stores the data needed by the task in a memory buffer, and then notifies the corresponding chip's FXCC, passing a pointer to the buffer and to the code to execute. Furthermore, the PHost must then poll the FXCC to ensure the task spawn has been successful, and repeat the operation if this has not been the case. In many cases, these tasks are spawned on the same chip and they execute the same code on different data. Consequently, we could significantly reduce the overhead by adding special hardware in the FXCC. That hardware could spawn the multiple tasks that execute the same code on the same chip in a single action. It could also queue the tasks that cannot be spawn immediately because the corresponding PArray is busy. The PHost would only have to fill the several buffers and then pass, with a single message, an array of buffer pointers and a pointer to the code. Moreover, if the buffers are placed in memory with a fixed stride, the PHost would only need to pass a single buffer pointer that points to the first buffer and the stride. Of course, in all cases, if the tasks need to execute different codes, the PHost would have to pass an array of code pointers.

We have simulated the optimization where the PHost passes to an FXCC a single buffer pointer, the stride, the number of buffers, and a single code pointer, and the FXCC then spawns all the tasks. The FXCC latency of the actual spawn does not change: it is the sum of the FXCC latencies of all the spawns. We call this optimization M for *Multi-spawn*, and apply it to the *Opt+H+C+L* versions of the codes to obtain *Opt+H+C+L+M*.

Figure 11 shows the speedups of all the applications for *Opt+H+C+L+M*. The figure also includes the original speedups of Figure 9 (*Opt*) and *Opt+H+C+L*. The H, C, and L optimizations mostly affect Swim and Mgrid. Overall, the figure shows that the multi-spawn optimization significantly speeds up Path and, to a lesser degree, TreAdd. Consequently, it is a useful improvement.

In summary, with all the software optimizations, a FlexRAM system with 1 or 2 FlexRAM chips delivers an average speedup of 10 and 11, respectively, over a conventional server. If we include the multi-spawn hardware optimization, the average speedups become 11 and 12, respectively.

8.4 Other Optimizations

Recall from Table 3 that we deliberately minimized the number of directives and additional lines added to our codes. In this final experiment, we show that high-speedup versions of the applications can be written with additional code tuning. For example, we have developed a version of Dmxdm with 148 lines of code and 12 CFlex directives. This is in contrast to the version used up until now, which has 81 lines and 1 directive. The tuned version chooses a leader PArray in each FlexRAM chip to manage the movements of data to and from other FlexRAM chips. The other PArrays in the chip synchronize with the leader using FXCC locks, before consuming the data from the leader's buffer. The resulting code doubles the speedup of Dmxdm in Figure 11 for two chips. Overall, therefore, much higher speedups than those in Figure 11 can be obtained if we are willing tune the codes.

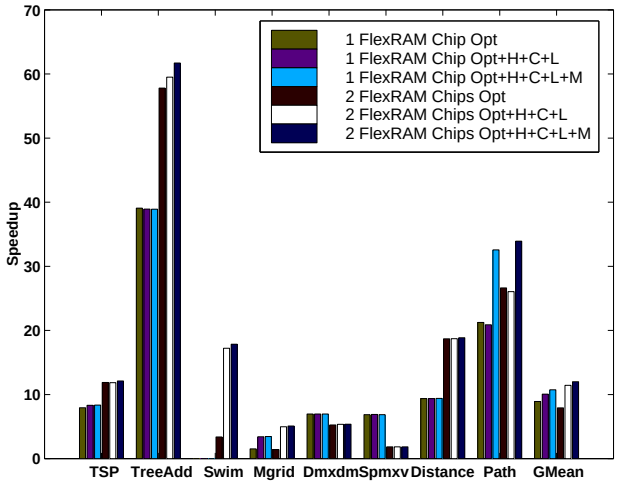

Figure 11. Impact of compiler, run-time, and hardware optimizations for task spawning and mapping.

9. RELATED WORK

Some proposals for programming intelligent memories [6, 10, 15] force the programmer to directly manage low-level operations such as communication via messages, cache management, data layout, or computation and data collocation. As a result, programming these machines is hard. Other proposals use a compiler to automatically partition the code into sections and then schedule each section on the appropriate processor [2, 17]. However, this approach has only been tried for simple codes and architectures.

More recently, Gilgamesh [19] proposes a middleware to simplify the programming of a machine with homogeneous, processor-memory nodes. This middleware supports object-based management, allowing locality and load balancing with dynamic control. Some differences between Gilgamesh and CFlex are that Gilgamesh lacks support for heterogeneous processors, it targets a different system (a highly-parallel NUMA) and, because of the size of the expected system, it tolerates and requires major code re-writing.

Let us consider now OpenMP [14] and HPF [8], the most widely-known parallelization directives. CFlex is inspired by OpenMP. There are, however, several important differences. One is that the OpenMP machine model is UMA and, as a result, OpenMP lacks the locality-related clauses found in CFlex. As for HPF, local memories are meaningful to HPF, but it uses replication and alignment to take advantage of them. While this strategy may be adequate for relatively regular codes, data structures enabled by C pointers and structs, which are the focus for our work, cannot be partitioned with these directives. Note that CFlex provides mechanisms for implicitly distributing and even aligning both regular and irregular data structures (Section 4.4).

Task definition and synchronization is also more powerful in CFlex than in OpenMP or HPF. The `sync/async` clauses enable the spawn of new tasks dynamically outside loops. In

this way, it is possible to parallelize recursive algorithms and the processing of lists, trees, and other pointer-based structures using our family of directives. This is particularly interesting for FlexRAM, as intelligent memory architectures are particularly well suited for codes with irregular access patterns. This gives CFlex a very important advantage over OpenMP, which can only parallelize iterative constructs of the `for/do` type and the statically nested parallelism of the `sections` and `section` directives. HPF is also primarily designed to exploit loop level parallelism, although version 2.0 [8] includes the `TASK_REGION` directive, which enables it to implement parallel sections, nested parallelism, and data-parallel pipelines. Still, it is less powerful than CFlex, as the generated tasks have several restrictions. For example, all the data that they access must be mapped locally to the active processor subset.

10. CONCLUSIONS

We presented an environment that enables the effective hand programming of intelligent memory architectures at a high level and with very modest effort. We used FlexRAM as a prototype architecture, but our approach is suitable for other, similar architectures. Our proposal is based on CFlex, a family of high-level compiler directives inspired by OpenMP. The CFlex environment gives the programmer high-level control over the assignment of computation to the main and memory processors, the layout of the data, and the synchronization between processors in a single-address space. We also proposed the use of IMO libraries. These functions program the processors in memory through CFlex, but make the processors transparent to the programmer.

Our experiments showed that CFlex and IMOs enable the low-effort generation of well-tuned parallel programs. Specifically, a set of conventional programs with 240 lines on average were transformed into CFlex parallel form with only 7 CFlex directives and 2 additional statements on average. Moreover, programs run on a single-chip FlexRAM system about 10 times faster on average than on a conventional server. Still, further improvements may be obtained by careful optimization of the codes, which suggests that the use of IMOs needs to be explored more.

11. REFERENCES

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Press, 1994.

[2] J. Chame, J. Shin, and M. Hall. Code Transformations for Exploiting Bandwidth in PIM-Based Systems. In *Solving the Memory Wall Problem Workshop*, June 2000.

[3] K. Diefendorff. Power4 Focuses on Memory Bandwith. *Microprocessor Report*, 13(13), October 1999.

[4] C. Foster. *Content Addressable Parallel Processors*. Van Nostrand Reinhold Co, New York, 1976.

[5] B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. CFlex: A Programming Language for the FlexRAM Intelligent Memory Architecture. Technical Report UIUCDCS-R-2002-2287, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2002.

[6] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping Irregular Applications to DIVA, a PIM-Based Data-Intensive Architecture. In *Supercomputing*, November 1999.

[7] M. Hall and C. Steele. Memory Management in PIM-Based Systems. In *Proceedings of the Workshop on Intelligent Memory Systems*, November 2000.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 2.0. 1997.

[9] S. Iyer and H. Kalter. Embedded DRAM Technology: Opportunities and Challenges. *IEEE Spectrum*, 36(4):56–64, April 1999.

[10] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design*, pages 192–201, October 1999.

[11] P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies. In *Proceedings of the 1996 Frontiers of Massively Parallel Computation Symposium*, 1996.

[12] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, September 1997.

[13] K. Mai, T. Paaske, N. Jayasena, R. Ho, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[14] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface Version 2.0. March 2002.

[15] M. Oskin, F. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *International Symposium on Computer Architecture*, pages 192–203, June 1998.

[16] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[17] Y. Solihin, J. Lee, and J. Torrellas. Automatic Code Mapping on an Intelligent Memory Architecture. *IEEE Transactions on Computers*, 50(11):1248–1266, November 2001.

[18] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[19] T. Sterling and H. Zima. The Gilgamesh MIND Processor-in-Memory Architecture for Petaflops-Scale Computing. In *4th International Symposium on High Performance Computing*, pages 1–5, 2002.

[20] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it All to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.

[21] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[22] S. Yoo, J. Renau, M. Huang, and J. Torrellas. FlexRAM Architecture Design Parameters. Technical Report CSRD-1584, Department of Computer Science, University of Illinois at Urbana-Champaign, October 2000. http://iacoma.cs.uiuc.edu/papers.html.